1. Connectionism

Perceptron

Inputs $\mathbf{x} \in \mathbb{R}^n$, Weights $\theta \in \mathbb{R}^n$. Threshold unit $(\mathbf{x}, \theta) = \text{sign}(\mathbf{x} \cdot \theta) = 1$ if $\mathbf{x} \cdot \theta \geqslant 0$.
Perceptron update: $\Delta\theta = 0$ if y sign$(\mathbf{x} \cdot \theta \geqslant 0)$, else yx. Often zigzags.
Weight-norm growth: Let$(\mathbf{x}^t, \mathbf{y}^t)$ be a sequence of perceptron mistakes, where $\theta^s = \sum_{t=1:s} \Delta\theta^t$, Then $|\theta^s|^2 \leqslant \sum_{t=1:s} |x^t|^2$, proof given by update rule.
Convergence theorem: The perceptron performs at most s $\leqslant |\frac{R}{\gamma}|^2$ update steps on any γ-separable (margin) and R-bounded sample S. proof start with $\gamma s \leqslant \sum_{t=1:s} \Delta\theta^t \cdot \theta^* \leqslant R\sqrt{s}$
linear Dichotomies: s points, C(s, n) = min C(S, n) = $2\sum_{i=1:n-1} \binom{s-1}{i}$ where $|S| = s$. Count how many dichotomies are possible with linear separators. C(s+1, n) = C(s, n) + C(s, n-1) (Cover's theorem). C(n, n) $= 2^n$. C(n+1, n) $= 2^{n+1} - 2$ If s>2n, almost all dichotomies are not linearly realizable. All dichot are linear if s<n

Hopfield Networks

Willshaw Memory: Sparseness should ideally scale as $n \propto \lg N$, assume design control over n.
Binary neurons : $x_i \in \{-1, 1\}$, couplings $\Theta \in \mathbb{R}^{N*N}$ 1) symmetric, 2) diagonal elements are 0. Recurrent dynamics: $x_i = \text{sign}(\sum_{j \neq i} \theta_{ij} x_j)$. If the weight matrix does not contain a zero diagonal, the network dynamics do not necessarily lead to stable states.
$\Theta = \mathbf{x}\mathbf{x}^\top - \mathbf{I}$, so $\Theta\tilde{\mathbf{x}} = (N - 2n - 1)\tilde{\mathbf{x}}$, if $n < N/2$ we retrieve $\mathbf{x}$ else $-\mathbf{x}$. Maximum Capacity: s $\leqslant$ 2N.

MLP

f(x;β,θ) $= \sum_{j=1}^{M} \frac{\beta_j}{1+exp(-\theta_j \cdot x)}$, parameters are β and θ.
Loss function with respect to β and θ: $\frac{\partial \frac{1}{2}(f(x)-y)^2}{\partial \beta_j} = \frac{f(x)-y}{1+exp(-\theta_j \cdot x)}$, $\frac{\partial \frac{1}{2}(f(x)-y)^2}{\partial \theta_{ji}} = \frac{f(x)-y}{1+exp(-\theta_j \cdot x)} \cdot \frac{-\beta_j x_i}{1+exp(\theta_j \cdot x)}$

2. Linear Networks

Linear Units

$\mathbf{x} \cdot \theta = \sum_{i=1}^{n} x_i \theta_i$
$\zeta(\mathbf{x}, \theta, \text{b}) = \mathbf{x} \cdot \theta + \text{b} = \binom{x}{1} \cdot \binom{\theta}{b}$
Linear units defines: 1) direction of change via $\theta/|\theta|$ 2) rate of change via $|\theta|$. However, constant rate of change is too restrictive.
Delta rule: $\Delta\theta = \eta\ (y - x \cdot \theta)$ x, gradient of squared loss.
homogeneity: f(αx)= αf(x), **additivity** : f(x+y) =f(x) +f(y) if f is linear.
Compositionality:let g, f be linear functions, then g $\circ$ f is linear. Proof by the above two properties. It does generalize to affine functions.

Linear Autoencoder

let $x \mapsto z \mapsto y, z = Cx, y = Dz$, def $l = \frac{1}{2}||x - y||^2 = \frac{1}{2}||x - DCx||^2$. Goal: $DC \approx \mathbb{I}$.
Gradients:
$\frac{\partial l(x)}{\partial C_{ki}} = \sum_{j=1}^{n}(y_j - x_j)\frac{\partial y_j}{\partial C_{ki}} = x_i \sum_{j=1}^{n} D_{jk}(y_j - x_j) = \mathbf{D}^\top(\mathbf{y} - \mathbf{x})\mathbf{x}^\top \in \mathbb{R}^{m \times n}$
$\frac{\partial l(x)}{\partial D_{jk}} = \sum_{i=1}^{n}(y_i - x_i)\frac{\partial y_i}{\partial D_{jk}} = (y_j - x_j)\sum_{i=1}^{n} C_{ki}x_i = (\mathbf{y} - \mathbf{x})\mathbf{x}^\top\mathbf{C}^\top \in \mathbb{R}^{n \times m}$
Rank Constraint: rank(DC)$\leqslant$ min{rank(C), rank(D)} $\leqslant$ m $\leqslant$ n $\Rightarrow \mathbf{Y} = \mathbf{DCX}$, rank($Y$) $\leqslant$ m.

SVD: $X = U\Sigma V^T$, where $X \in \mathbb{R}^{n \times s}$, $U \in \mathbb{R}^{n \times n}$, $V \in \mathbb{R}^{s \times s}$, orthogonal, $\Sigma = \text{diag}(\sigma_1, ..., \sigma_{\min\{n,s\}})$
TruncatedSVD: $X_r = U_r\Sigma_r V_r^T$, which induced Eckart-Young Theorem: $|X - X_r|_F = \min_{rank(Y) \leqslant r} |X - Y|_F$
How to choose C and D to be optimal: $C = U_m^T$ and $D = U_m$

Deep Linear Gradients

inputs X, targets Y. Assum X and Y are centered and X is whitened: $\frac{1}{s}XX^T = U\Lambda U^T$, $X \mapsto \Lambda^{-\frac{1}{2}}U^T X$, the least square problem: $\frac{1}{2}|\Theta - \frac{1}{s}YX^T|_F^2$
Therefore the autoencoder loss will become: $\frac{1}{2}|DC - \frac{1}{s}YX^T|_F^2$
$\frac{\partial l}{\partial C} = D^T(DC - \frac{1}{s}YX^T)$, $\frac{\partial l}{\partial D} = (DC - \frac{1}{s}YX^T)C^T$
changebases: such that Γ (DC) becomes diagonal, since different directions decouple. Let $\Gamma = U\Sigma V^T$, DC - $\Gamma = U(U^T DCV - \Sigma)V^T = U(\tilde{D}\tilde{C} - \Sigma)V^T$, where $\tilde{D} = U^T D$ and $\tilde{C} = CV$. Diagonalized form: $\frac{\partial l}{\partial \tilde{C}} = (\tilde{D}\tilde{C} - \Sigma)^T\tilde{D}$, $\frac{\partial l}{\partial \tilde{D}} = \tilde{C}(\tilde{D}\tilde{C} - \Sigma)^T$.
As time progresses, the mode will decouple.

3. Sigmoid Networks

Ridge Functions

Idea: compose linear function with a non-linear function Φ,f = $\Phi(\mathbf{x} \cdot \theta)$. It preserves the directional sensitivity of linear functions. It has a variable rate of change, governed by Φ'. Loss function l w.r.t. θ gradient: $\Delta_\theta(l \circ \phi)(\mathbf{x} \cdot \theta) = (l \circ \phi)'(\mathbf{x} \cdot \theta)\mathbf{x}$

Threshold Units

Sign units /Heavyside (Specified in 1. Connectionism)
Drawbacks: the function has a jump at 0 1) no derivative information 2)rule out gradient based methods. **Alternative**: used in perceptron learning.
PerceptronLearning: $\mathcal{L}(\mathbf{x}, y; \theta) = \max\{0, -y\mathbf{x} \cdot \theta\}$

Sigmoid Units

$\sigma(\mathbf{x} \cdot \theta) = \frac{1}{1+exp(-\mathbf{x} \cdot \theta)}$ defines a smooth and monotone function. **Derivatives**: $\sigma'(z) = \sigma(z)(1 - \sigma(z)) = \sigma(z)\sigma(-z)$. Derivatives are polynomials in σ
Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1$. Its symmetric around 0.
Inverse: $\sigma^{-1}(t) = ln(\frac{t}{1-t})$
LogisticRegression: done with a sigmoid unit. Cross-entropy loss: $\mathcal{L}(\mathbf{x}, y; \theta) = -ln(\sigma(y\mathbf{x} \cdot \theta)) = -ln(P(Y = y|\mathbf{x}, \theta))$. when using {0,1} encoding, this is equivalent to $-yln\sigma(\mathbf{x} \cdot \theta) - (1 - y)ln(1 - \sigma(\mathbf{x} \cdot \theta))$
LogisticSGD: $-\Delta_\theta l(\mathbf{x}, y) = \sigma(-y\mathbf{x} \cdot \theta)y\mathbf{x}$

Softmax

$\sigma_i^{max}(\mathbf{x}, \Theta) = \frac{exp(\mathbf{x} \cdot \theta_i)}{\sum_{j=1}^{k} exp(\mathbf{x} \cdot \theta_j)}$, strictly positive. sum over i is 1.
Gradient:
$\nabla_{\mathbf{x}_j}\text{softmax}(\mathbf{x})_i = \text{softmax}(\mathbf{x})_i(1 - \text{softmax}(\mathbf{x}))_i$ if $i = j$, $-\text{softmax}(\mathbf{x})_i \cdot \text{softmax}(\mathbf{x})_j$ if $i \neq j$
Softmax + Cross-entropy: $l(\mathbf{x}, \mathbf{y}; \Theta) = -\mathbf{y} \cdot \ln\sigma^{\max}(\mathbf{x}, \Theta)$. Alternatively separate out the normalization: $l(\mathbf{x}, y; \Theta) = -\sum_{i=1}^{k} y_i \cdot x \cdot \theta_i + \ln(\sum_{i=1}^{k} \exp[\mathbf{x} \cdot \theta_i])$
Gradient: $\nabla_{\theta_i} l(\mathbf{x}, y; \Theta) = (\sigma_i^{\max} - y_i)\mathbf{x}$

4. Approximation Theory

Weierstrass Theorem

Polynomials are dense in C(I), where I = [a;b] for any a<b. It could be also stated as: given a f(x) for a$\leqslant$x$\leqslant$b and if ϵ is an arbitrary positive quantity, it is possible to construct an approximating polynomial P(x) such that $|f(x) - P(x)| \leqslant \epsilon$
Proof: Bernstein basis polynomials: $b_k^m(x) = \binom{m}{k}x^k(1 - x)^{m-k}$ degree m.
$q_m(x) = \sum_{k=0}^{m} f(\frac{k}{m})b_k^m(x)$.. Consider residuals $|f(x) - q_m(x)| = |\sum_{k=0}^{m} r_k^m(x)|$, where $r_k^m(x) = [f(x) - f(\frac{k}{m})]b_k^m(x)$. 3) Upper bound by splitting I : $k : |x - \frac{k}{m}| \leqslant \delta$ and I^C, δ is chosen such that $|f(x) - f(y)| \leqslant \epsilon/2$. And thus $\sum_{k \in I} r_k^m(x) \leqslant \epsilon/2$. By concentration: $\sum_{k \notin I} r_k^m(x) \leqslant R\sum_{k \notin I} b_k^m(x) \approx 0$ when m to ∞.

approximation

if the smooth function σ is not a polynomial, H = span$g : g = \sigma$) is a universal approximator. proof: calculate the derivatives, so the derivatives are all polynomials of σ. If linear then overall linear.
n-layers with non-linear activation is a universal function approximator.
Spans of ridge functions are universal approximators.

Complexity

Fourier transformation of gradient function has to be absolutely integrable.
MLP does not suffer from curse of dimensionality.
Freedom in the choice of data distribution
Remarkable approximation error rate $\propto 1/m$
Linear combination of m basis functions has lower approx. erro bound $(\propto (1/m)^{2/n})$
Benefits of depth: 1) Deeper yields better. 2) Idea: radial function

5. Backpropagation

Layers

Feedforward networks: Directed Acyclic Graphs (DAGs) represent connectivity structure between computational units.
Width of a layer: $\text{width}_l = n_l := \dim(\text{range}(F_l))$
Total **number of units** in the networks is:
$\#\text{units} = \sum_{l=1}^{n} \text{width}_l$
activations: Layer-to-layer maps: successively transform input into intermediate representations.

Multivariate Analysis

Jacobian: Generalization to multivariate maps $F : \mathbb{R}^n \to \mathbb{R}^m$
$\partial_{ij}F = \partial_i F_j : \mathbb{R}^n \to \mathbb{R}$, $\partial F = (\partial_{ij}F) : \mathbb{R}^n \to \mathbb{R}^{m \times n}$
where ∂F is the Jacobian map of F (rows of Jacobian are gradients of component functions of F).
Chain Rule:
$\partial F = \prod_{1}^{l=k} \partial F_l \circ \partial F_{l-1:1}$ or $\partial F(\mathbf{x}) = \prod_{1}^{l=k} \partial F_l(\mathbf{z}_{l-1})$

Gradient Descent

Loss Function: Loss function: a measure of approximation quality. $\hat{\mathbf{y}}$: model prediction vs. $\mathbf{y}$: ground truth.

Backpropagation

Goal: Want to learn the parameters θ so that we minimize a given loss function.
Loss Function Consider the loss $f(x) = \frac{1}{2}\|F(\mathbf{x}) - \mathbf{y}\|_2^2$ function where $\mathbf{y}$ is a target vector.

6. Rectified Networks

Rectified linear unit (ReLU)

def $\to (\mathbf{x}, \theta) \mapsto (\mathbf{x} \cdot \theta)_+ = \max\{0, \mathbf{x} \cdot \theta\}$
Layer of *m* ReLU units on fixed input $\mathbf{x}$: each unit will either be active or inactive.
Benefit of the ReLU: gradient norm does not vanish due to saturation.
Networks with one hidden layer of ReLU or absolute value units are universal function approximators.

Absolute Value Unit

$|z| := \begin{cases} z & \text{if } z \geqslant 0 \\ -z & \text{otherwise} \end{cases}$
Relation to ReLU activation: $(z)_+ = \frac{z+|z|}{2}$

Smooth ReLU Approximations

Softplus: $(\mathbf{x}; \theta) \mapsto \ln(1 + \exp[\mathbf{x} \cdot \theta]) \in (0; \infty)$
Exponential linear unit:
$(\mathbf{x}; \theta) \mapsto \begin{cases} \mathbf{x} \cdot \theta & \text{if } \mathbf{x} \cdot \theta \geqslant 0 \\ \exp[\mathbf{x} \cdot \theta] - 1 & \text{else} \end{cases} \in (-1; \infty)$
Leaky ReLU: patches-up ReLUs to avoid lack of gradient information in the zero branch (typical $\epsilon = 0.01$)
$(\mathbf{x}; \theta) \mapsto \begin{cases} \mathbf{x} \cdot \theta & \text{if } \mathbf{x} \cdot \theta \geqslant 0 \\ \epsilon\mathbf{x} \cdot \theta & \text{else} \end{cases} \in \mathbb{R}$

Hinge Functions

Hinge function: 2-armed ReLUs def: g(x) = max($\theta_1 \cdot x + b_1, \theta_2 \cdot x + b_2$). 2max(f, g) = f + g + $|f - g|$. Generalized to k-hinge function: g(x) = max$\{\theta_j \cdot x + b_j\}$. Every continuous piecewise linear function g can be written as a signed sum of k-Hinges with k$\leqslant$n+1. (exact, Maxout)
Polyhedral S Polyhedral, S is finite intersection of closed half-spaces. Every continuous piecewise linear function f can be written as the difference of two polyhedral functions.$\max_{(\theta,b) \in \mathcal{A}^+}\{\theta \cdot x + b\} - \max_{(\theta,b) \in \mathcal{A}^-}\{\theta \cdot x + b\}$
Maxout nets with two maxout units are universal function approximators. Proof: 1) the last theorem 2) PWL functions are dense in C($\mathbb{R}^n$)

7. Optimization

Newton's method

Consider local approximation of f:
$f(x + \Delta x) \approx f(x) + \nabla f(x) \cdot \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^\top\nabla^2 f(\mathbf{x})\Delta\mathbf{x}$
Minimize over $\Delta\mathbf{x}$ is: $\Delta\mathbf{x} = -[\nabla^2 f(\mathbf{x})]^{-1}\nabla f(\mathbf{x})$
It requires computing and inverting the Hessian (expensive in high dimensions).
Preconditioning: Can we choose a different matrix $\mathbf{B}$ that is close to the Hessian (i.e. $||\nabla^2 f(\mathbf{x}) - \mathbf{B}||_2 \leqslant \epsilon$)
Diagonal approximation:$\mathbf{B} = \text{Diag}(\nabla^2 f(\mathbf{x}))$

Smoothness and Convexity

L-smooth function (L-Lipschitz-continuous gradient):
$||\nabla f(\mathbf{x}) - \nabla f(\mathbf{x} + \Delta\mathbf{x})|| \leqslant L||\Delta\mathbf{x}||$
μ-strongly convex function:
$f(\mathbf{x} + \Delta\mathbf{x}) \geqslant f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \Delta\mathbf{x} + \frac{\mu}{2}||\Delta\mathbf{x}||^2$, convex if $\mu = 0$.
Theorem 1: For a μ-strongly convex, L-smooth function l, the gradient descent iterates $\mathbf{x}^k$ with step size $0 < \eta < 1/L$ converges to the unique minimizer $\mathbf{x}^*$ at rate:
$||\mathbf{x}^k - \mathbf{x}^*||^2 \leqslant (1 - \eta\mu)^k||\mathbf{x}^0 - \mathbf{x}^*||^2$

PL condition: generalization of strong convexity without the convexity: def $\rightarrow$ $\frac{1}{2}||\nabla f(\mathbf{x})||^2 \geqslant \mu(f(\mathbf{x}) - f^*)$, $\forall \mathbf{x}$, $f^* = \min f(\mathbf{x})$
Theorem 2: Let f be differentiable and L-smooth, not necessarily convex with minimum f^* and fulfilling the PL condition with $\mu > 0$. The gradient descent iterates with step size $\eta \leqslant 1/L$ satisfy
$f(\mathbf{x}^k) - f^* \leqslant (1 - \frac{\mu}{L})^k (f(\mathbf{x}^0) - f^*)$

Stochastic Gradient Descent

def $\mapsto$ $\mathbf{x}^{k+1} = \mathbf{x}^k - \eta_k \nabla f_{I(k)}(\mathbf{x}^k)$, $I(k) \sim \text{Uniform}\{1, ..., n\}$
Update direction of SGD is unbiased. We need to control the variance term in order to ensure convergence. 1) Averaging iterates; 2) Use appropriate decreasing step-size.
Minibatch SGD: unbiased update, variance reduced $\propto$ r (training instances in each update)

Momentum and Adaptivity

Polyak's Heavy Ball Method:
$\mathbf{x}^{k+1} = \mathbf{x}^k - \eta \nabla f(\mathbf{x}^k) + \beta(\mathbf{x}^k - \mathbf{x}^{k-1})$, $\beta \in (0; 1)$
Nesterov's Accelerated Method:
$\mathbf{y}_{k+1} = \mathbf{x}_k + \beta(\mathbf{x}_k - \mathbf{x}_{k-1})$, $\mathbf{x}_{k+1} = \mathbf{y}_{k+1} - \eta \nabla f(\mathbf{y}_{k+1})$
AdaGrad:
$\gamma^k = \gamma^{k-1} + \nabla f(\mathbf{x}^k) \odot f(\mathbf{x}^k)$, where $\odot$ denotes pointwise multiplication. γ_i^k corresponds to the sum of the squares of the i-th parameter's partial derivatives along the iterates $\mathbf{x}^0, ..., \mathbf{x}^k$. $\mathbf{x}^{k+1} = \mathbf{x}^k - \eta \Lambda^k \nabla f(\mathbf{x}^k)$, where $\Lambda^k = \text{diag}(\lambda_i^k)$, with $\lambda_i^k = 1/(\sqrt{\gamma_i^k} + \delta)$, $\delta > 0$
Adam: Diagonal pre-conditioner with momentum term
$\mathbf{x}^{k+1} = \mathbf{x}^k - \frac{\eta}{1-\beta} \Lambda^k \mathbf{m}^k$, $\Lambda^k = \text{diag}(\lambda_i^k)$, with $\frac{1}{\lambda_i^k} = \sqrt{\frac{\gamma_i^k}{1-\alpha}} + \delta$, typically choices are $\beta = 0.9$ and $\alpha = 0.999$
AMSGrad: fixes a problem that may prevent Adam from converging (even on convex functions).

8. Regularization

Motivation

Regularization: Any aspect of a learning algorithm that is intended to lower the generalization error but not the training error.
Informed regularization: encode specific **prior knowledge**; Simplicity bias: preference for **simpler** models; Model averaging: e.g. **ensemble** methods, drop-out; Data augmentation and cross-task learning.
Norm-based Regularization:
1. Standard regularization method (convex models):
$\mathbf{R}_\Omega(\theta; S) = \mathbf{R}(\theta; S) + \Omega(\theta)$
S: sample of training data
Ω: functional that does not depend on training data
2. L_2/Frobenius–norm penalty for deep networks:

$$\Omega(\theta) = \frac{1}{2} \sum_{l=1}^{L} \mu^l \|\Theta^l\|_F^2, \mu^l \geqslant 0$$

common practice: only penalize weights, not biases.
single μ weight or one μ^l per layer (in the sequel: single μ).

Weight Decay

1. L_2-regularization: also called weight decay as
$\nabla\Omega = \mu\theta$ or $\frac{\partial\Omega}{\partial\theta_{ij}^l} = \mu\theta_{ij}^l$
weights get pulled towards zero with "gain" μ.
naturally favors weights of small magnitude.
2. Gradient descent gets modified as
$\theta(k+1) = (1 - \mu\eta)\theta(k) - \eta\nabla R(\theta(k); S)$
3.Along directions in parameter space with large eigenvalues of $\mathbf{H}$, i.e. $\lambda_i \gg \mu$: **vanishing effect**.
4.Along directions in parameter space with small eigenvalues of $\mathbf{H}$, i.e. $\lambda_i \gg \mu$: **shrunk** to nearly zero magnitude.
5. Quadratic (Taylor) approximation of R around optimal $\theta^\star$:
$R(\theta) \approx R(\theta^\star) + \frac{1}{2}(\theta - \theta^\star)^\top \mathbf{H}(\theta - \theta^\star)$
where $\mathbf{H} := (\frac{\partial^2 R}{\partial\theta_i \partial\theta_j})|_{\theta=\theta^\star}$
6. First order **optimality condition**:
$\nabla R_\Omega \overset{!}{=} 0 \Leftrightarrow \mathbf{H}(\theta - \theta^\star) + \mu\theta = 0$
$\Leftrightarrow (\mathbf{H} + \mu\mathbf{I})\theta = \mathbf{H}\theta^\star \Leftrightarrow \theta = (\mathbf{H} + \mu\mathbf{I})^{-1}\mathbf{H}\theta^\star \Leftrightarrow \mathbf{Q}(\Lambda + \mu\mathbf{I})^{-1}\Lambda\mathbf{Q}^\top\theta^\star$
with diagonalisation $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$, $\Lambda = \text{diag}(\lambda_1, ..., \lambda_d)$.
7.Perform analysis exactly for special case: **linear regression**: $\theta = (\mathbf{X}^\top\mathbf{X} + \mu\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$, which is called **ridge regression**.
$\mu \rightarrow 0$: Moore-Penrose pseudoinverse

Early Stopping

Early stopping: stop learning after finite (small) number of iterations.
Rely on **validation data** (hold-out) to estimate and track expected risk. Stop when flat/worsening. Keep best solution.
Early stopping acts as an approximate L_2-regularizer.
Euqating coefficients: $k = \frac{1}{\eta\mu} \Leftrightarrow k\eta = \frac{1}{\mu}$

Ensemble Methods

Bagging: Ensemble method that combines models trained on bootstrap samples.
bootstrap sample S_r^k sample r times from S **with replacement** for $k = 1, ..., K$, (many duplicates, on average $\approx 2/3$ distinct examples)
train model on $S_r^k \rightarrow \theta^k$.
prediction: average model output probabilities $p(\mathbf{y}|\mathbf{x}; \theta^k)$

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{K} \sum_{K}^{k=1} p(\mathbf{y}|\mathbf{x}; \theta^k)$$

There are many ways of creating ensembles.
training data randomization (see above)
stochastic model training (initialization, SGD)
different architectures and model classes
Almost always beneficial, but expensive (memory, compute).

Knowledge Distillation

Idea: Transfer knowledge from a complex model (source) into a simpler one (target).
Goals: performance, memory/energy efficiency, understandability.
Particularly interesting with ensembles of DNNs.
Blackbox distillation:
1. use unlabeled data and run complex model to label it.
2. train target model on augmented training data.
3. refinement: use soft targets.

DropOut

Dropout idea: randomly "drop" subsets of units in network.
More precisely, define "keep" probability π_i^l for unit i in layer l.
typically: $\pi_i^0 = 0.8$ (inputs) and $\pi_i^{l \geqslant 1} = 0.5$ (hidden units).
realization: sampling bit mask and zeroing out activations.
effectively defines an exponential ensemble of networks (each of which is a sub-network of the original one).
all models share same weights.
standard backpropagation applies.
Dropout realizes an ensemble $p(\mathbf{y}|\mathbf{x}) = \sum_{\mathbf{Z}} p(\mathbf{Z})p(\mathbf{y}|\mathbf{x} : \mathbf{Z})$
where $\mathbf{Z}$ denotes the binary "zeroing" mask (note that $p(\mathbf{Z})$ is defined via probabilities π_i^l)
Prediction: no analytic solution for deep networks known. Practically: sample $\mathbf{Z}$ and average results ($\approx 10 - 20$ repetitions)
Weight Rescaling: Simple, effective heuristic (to avoid 10-20x sampling blowup): scale each weight θ_{ij}^l by probability of unit j being active, $\tilde{\theta}_{ij}^l \leftarrow \pi_j^{l-1}\theta_{ij}^l$
Makes sure, net input to unit i is calibrated, i.e.
$\sum_j \tilde{\theta}_{ij}^l x_j \overset{!}{=} \mathbf{E}_\mathbf{z} \sum_j z_j^{l-1}\theta_{ij}^l x_j = \sum_j \pi_j^{l-1}\theta_{ij}^l x_j$

Data Augmentation

Often one knows a priori that inputs can be subjected to certain transformations τ without changing the target output. Input **invariances** (global or local).
Injection of Noise: Various schemes to inject noise during training: 1) adding noise to **inputs**: ideally realistic noise (e.g. background noise in acoustic or image processing). 2) adding noise to **weights**: regularizing effect – find weights where small perturbations have small effects on outputs. 3) adding noise to **targets**: soft targets (e.g. probability distributions over labels, robustness w.r.t. label errors)

Task Augmentation

Semi-supervised Learning: Typical setting: more unlabeled data than labeled data. Define a generative model with a corresponding log-likelihood. Optimize **additive combination** of supervised and unsupervised risk, while **sharing parameters**. Generally more effective (but also more expensive) than pre-training.
Pre-Training: Pre-train parts of the model on a more generic task. Can also **fine-tune** parameters, i.e. initialize with pre-trained parameters
Multi-Task Learning: Share representations across tasks and learn jointly (i.e. minimize combined objective). Typical architecture: share low level representations, learn high level representations per task. Sometimes even task-specific linear layers are sufficient.

9. Convolutional Networks

definition

Given two functions f,h, their convolution is defined as
def $\rightarrow$ $(f * h)(u) := \int_{-\infty}^{\infty} h(u - t)f(t)\,dt = \int_{-\infty}^{\infty} f(u - t)h(t)\,dt$
corresponds to an integral operator with kernel $H(u, t) = h(u - t)$ operating on f. By symmetry one can also think of $F(u, t) = f(u - t)$ operating on h.

Shift Equivariance

Shifted function $f_\Delta(t) := f(t + \Delta)$. Convolution is shift-equivariant, i.e. $\Rightarrow f_\Delta * h = (f * h)_\Delta$
$(f_\Delta * h)(u) = \int_{-\infty}^{\infty} h(u-t)f(t+\Delta)\,dt \overset{t\mapsto t-\Delta}{=} \int_{-\infty}^{\infty} h(u + \Delta - t)f(t)\,dt = (f * h)(u + \Delta)m = (f * h)_\Delta(u)$
The convolution operator is commutative, one can exchange the function and the kernels.

Discrete Convolution

In practice: signals (digitally) sampled. Need to consider discrete case.
Let $f, h : \mathbb{Z} \rightarrow \mathbb{R}$. Define discrete convolution via
def $\rightarrow$ $(f * h)[u] := \sum_{t=-\infty}^{\infty} f[t]h[u - t]$
typical choice of h: support over finite window, e.g. $h[t] = 0$ for $t \notin [t_{\min}; t_{\max}]$

Cross–correlation

Sibling to convolutions: cross–correlation. In the discrete case:
def $\rightarrow$ $(f \star h)[u] := \sum_{t=-\infty}^{\infty} f[t]h[u + t]$
sliding inner product; difference to convolution: $u + t$ instead of $u - t$; flipped over kernel: $(h \star f) = (\bar{h} * f)$, where $\bar{h}[t] := h[-t]$.

Convolutional Networks

Convolutions in Higher Dimensions: matrices or fields (e.g. in discrete case for 2D):

$$(F * G)[i, j] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} F[i - k, j - l] \cdot G[k, l]$$

tensors: for 3D and higher
Border Handling: 1) padding with zeros = **same padding**; 2) only retain values from windows fully contained in support of signal f = **valid padding**
Backpropagation: **Weight sharing** in computing $\frac{\partial R}{\partial h_j^l}$, where h_j^l is a kernel weight $\frac{\partial R}{h_j^l} = \sum_i \frac{\partial R}{\partial x_j^l} \frac{\partial x_j^l}{\partial h_j^l}$, weight is re-used for every unit within target layer $\Rightarrow$ additive combination of derivatives in chain rule.
Receptive Fields: nesting of convolutions: receptive fields grow.

Efficient Computations of Convolutional Activities

FFT (**Fast Fourier Transform**): compute convolutions fast(er). Fourier transform of signal $f \rightarrow (Ff)$ and kernel $h \rightarrow (Fh)$. pointwise multiplication and inverse Fourier transform $(f * h) = F^{-1}((Ff)(Fh))$.
FFT: signal of length n, can be done in $O(n \log n)$.
pays off, if many channels (amortizes computation of Ff).
small kernels ($m < \log n$): favor time/space domain.

Pooling

Define window size r (e.g. 3 or 3 $\times$ 3), then
1D: $x_i^{max} = \max\{x_{i+k} : 0 \leqslant k \leqslant r\}$
1D: $x_{ij}^{max} = \max\{x_{i+k,j+l} : 0 \leqslant k, l \leqslant r\}$
other functions are possible: average, soft-maximization.
Sub-Sampling (aka "Strides"): reduce temporal/spatial resolution; Disadvantage: loss of information.
Channels: Learn multiple convolution kernels (or filters) = multiple channels

Convolutional Layers: Properties

Pros: computationally efficient; (close to) translational equivariant; strong parameter sharing which keeps the number of trainable. parameters low
Cons: not all data are sequences or is translational equivariant; local receptive field which makes it hard to connect distant features.
The output size after CNN:

$$\left(\left\lfloor \frac{W_{in} + 2P - K}{S} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{H_{in} + 2P - K}{S} \right\rfloor + 1\right)$$

10. Deep Gradients

Short Connectivity

w/o residual connections: deeper networks is poorer.
w/ residual connections: advantage of depth.
General connectivity patterns:
Residual connections: shortcut layers and **add** back in.
Skip connections: shortcut layers and **concatenate** back in.
Common idea: add less deep paths to a very deep network

Normalisation

Batch Normalization:
Motivation: Hard to find suitable learning rate.
Key idea: normalize the layer activations $\Rightarrow$ batch normalization (and backpropagate through normalization!)
fix layer l, fix set of examples $I \subseteq [1:s]$
mean activity, vector of standard deviation
$\mu^l := \frac{1}{|I|} \sum_{t \in I} z^l[t], z^l[t] := (F^l \circ ... \circ F^1)(x[t])$
$\sigma_s^l := \sqrt{\delta + \frac{1}{|I|} \sum_{t in I} (z_i^l[t] - \mu_i^l)^2}, \delta > 0$
μ and δ are functions of the weights: can be differentiated.
Normalized activities (cf. z-score in statistics)
$\hat{z}_i^l := \frac{z_i^l - \mu_i^l}{\sigma_i^l}$, $\mathbf{E}[\hat{z}_i^l] = \mathbf{0}$, $\mathbf{E}[(\hat{z}_i^l)^2] = 1$
Regain representational power $\hat{z}_i^l = \alpha_i^l \hat{z}_i^l + \beta_i^l$, in principle: can exactly undo the batch normalization; however: different learning dynamics, mean activation (and scale) can now be directly controlled.
in Practice: Increase learning rate; Remove dropout; Decrease weight regularization; Accelerate learning rate decay.
(+): very effective, broadly used in vision
(-): dependence on batch size, not suitable for all architectures
Layer Normalisation: use population average of activations in layer as reference. Unlike batch normalization, layer normalization does not impose any constraint on the size of the mini-batch and it can be used in the pure online regime with batch size 1.
$\mu^l[t] := \frac{1}{m^l} \sum_{i=1}^{m^l} z_i^l[t]$
$\sigma^l[t] := \sqrt{\delta + \frac{1}{m^l} \sum_{i=1}^{m^l} (z_i^l[t] - \mu^l[t])^2}, \delta > 0$
$\mathbf{h}[t] = f_1(\frac{\mathbf{g}}{\sigma^l[t]} \odot (\mathbf{a}[t] - \mu[t]) + \mathbf{b})$
(+): very effective, used in natural language (transformers, RNNs)
(-): no consistent benefits

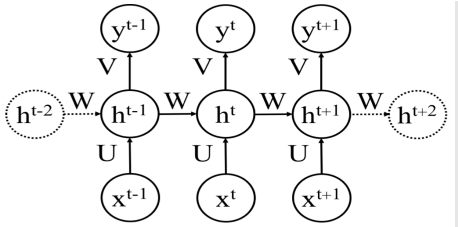
11. Recurrent Neural Networks

State Space Model

Given observation sequence $\mathbf{x}^1, ..., \mathbf{x}^s$. Identify hidden activities $\mathbf{h}$ with the state of a dynamical system. Discrete time evolution of **hidden state space sequence**.
$\mathbf{h}^t = F(\mathbf{h}^{t-1}, \mathbf{x}^t; \theta)$, $\mathbf{h}^0 = \mathbf{0}$, $t = 1, ..., s$
1. Markov property: hidden state at time t depends on input of time t as well as previous hidden state.
2. Time-invariance: state evolution function F is independent of time t.

Recurrent Neural Network

Motivation: Remove Markov assumption.
Linear dynamical system w/ elementwise non-linearity
$\bar{F}(\mathbf{h}, \mathbf{x}; \theta) = \mathbf{W}\mathbf{h} + \mathbf{U}\mathbf{x} + \mathbf{b}$, $\theta = (\mathbf{U}, \mathbf{W}, \mathbf{b}, ...)$
$F = \sigma \circ \bar{F}$, $\sigma \in$ {logistic, tanh, ReLU,...}
Optionally produce outputs via
$\mathbf{y} = H(\mathbf{h}; \theta)$, $H(\mathbf{h}; \theta) := \sigma(\mathbf{V}\mathbf{h} + \mathbf{c})$, $\theta = (..., \mathbf{V}, \mathbf{c})$



Sequence Loss Functions

1. output at the end ($t = T$: last step): $\vec{h}^T \mapsto H(\vec{h}^T; \theta) = \vec{y}$
Same as feedforward network: use loss J on $\mathbf{y}$.
2. output at every time step: sequence $\mathbf{y}^1, ..., \mathbf{y}^T$
Additive loss function:
$R(\mathbf{y}^1, ..., \mathbf{y}^T) = \sum_{t=1}^{T} R(\mathbf{y}^t) = \sum_{t=1}^{T} R(H(\vec{h}^t; \theta))$
Feedforward vs. Recurrent Networks: 1. For any fixed length s, the unrolled recurrent network corresponds to a feedforward network with s hidden layers.
2. However, inputs are processed in sequence and (optionally) outputs are produced in sequence.
3. Main difference: sharing of parameters between layers – same functions F and H at all layers / time steps.
Backpropagation through Time: Backpropagation is straightforward: propagate derivatives **backwards through time**.
Parameter sharing leads to sum over t, when dealing with derivatives of weights.
Define shortcut $\dot{\sigma}_i^t := \sigma'(\bar{F}_i(h^{t-1}, x^t))$, then
$\frac{\partial R}{\partial \omega_{ij}} = \sum_{t=1}^{s} \frac{\partial R}{\partial h_i^t} \cdot \frac{\partial h_i^t}{\partial \omega_{ij}} = \sum_{t=1}^{s} \frac{\partial R}{\partial h_i^t} \cdot \dot{\sigma}_i^t \cdot h_j^{t-1}$
$\frac{\partial R}{\partial u_{ij}} = \sum_{t=1}^{s} \frac{\partial R}{\partial h_i^t} \cdot \frac{\partial h_i^t}{\partial u_{ik}} = \sum_{t=1}^{s} \frac{\partial R}{\partial h_i^t} \cdot \dot{\sigma}_i^t \cdot x_k^t$
RNN Gradients: RNN where output is produced in the last step: $\mathbf{y} = \mathbf{y}^s$
Shared weights: $F^t = F$, yet evaluated at different points
$\nabla_{\mathbf{x}^t} R = [\prod_{r=t+i}^{s} \mathbf{W}^\top \mathbf{S}(\mathbf{h}^r)] \cdot \mathbf{J}_H \cdot \nabla_{\mathbf{y}} R$ where
$\mathbf{S}(\mathbf{h}^r) = \text{diag}(\dot{\sigma}_1^r, ..., \dot{\sigma}_n^r)$ which is $\leqslant \mathbf{I}$ for $\sigma \in$ {logistic, tanh, ReLU }

Bi-directional Recurrent Networks

Define **reverse order** sequence:
$\mathbf{g}^t = G(\mathbf{x}^t, \mathbf{g}^{t+1}; \theta)$ as model w/ separate parameters.
Backpropagation is also bi-directional.

Deep Recurrent Networks

hierarchical hidden state:
$\mathbf{h}^{t,1} = F^1(\mathbf{h}^{t-1,1}, \mathbf{x}^t; \theta)$,
$\mathbf{h}^{t,l} = F^1(\mathbf{h}^{t-1,l}, \mathbf{h}^{t,l-1}; \theta)$ $(l = 1, ..., L)$
Output connected to the last hidden layer $\mathbf{y}^t = H(\mathbf{h}^{t,L}; \theta)$

Memory Units

Motivation: difficult to learn long-term dependencies with standard recurrent network.
Long-Short-Term-Memory (LSTM): unit for memory management; Remembering information for long time and forgetting it fast. Four gates: i/o, forget/update. Solve the vanishing gradient problem (occurred in RNN).
Forget Gate: Keeping or forgetting of stored content?
$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
Input $\rightarrow$ Memory Value: Preparing new input information to be added to the memory
$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$ $\tilde{C}_t = \sigma(W_C \cdot [h_{t-1}, x_t] + b_C)$
Updating Memory: Combining stored and new information
$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
Output Gate: Computing output selectively
$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$ $h_t = o_t * \tanh(C_t)$
Gated Memory Units (GRU): Memory state = output; Convex combination of old and new information. 3 Gates which combines (Less than LSTM), so less training parameters.
$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$ $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$
$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$ $h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$
In GRU instead of the forget gate, we decide how much past information to retain or discard based on the compliment of the input gate vector.

Connectionist Temporal Classification

Simple model: $p(\pi|\vec{x}) = \prod_{t=1}^{T} y_{\pi_t}$ where π_t is a distribution over all possible labels + blank.
Map from many to one: $p(\mathbf{L}|\vec{x}) = \sum_{\pi \in B^{-1}(\mathbf{L})} p(\pi|\vec{x})$

Sequence-to-Sequence Learning

Goal: define conditional probability distribution over output sequence $\mathbf{y}^{1:T}$, given input sequence $\mathbf{x}^{1:T}$.
Naive implementation by RNN:
$\mathbf{x}^{1:t} \overset{F}{\mapsto} \mathbf{h}^t$, $\mathbf{h}^t \overset{H}{\mapsto} \mu^t$, $\mu^t \mapsto p(\mathbf{y}^t)$
Problem: $p(\mathbf{y}^t)$ depends on $\mathbf{y}^{1:t-1}$ only through $\mathbf{h}^t$.
Conditional independence assumption:
$p(\mathbf{y}^t|\mathbf{x}^{1:t}, \mathbf{y}^{1:t-1}) = p(\mathbf{y}^t|\mathbf{x}^{1:t})$
Seq2Seq Model: Encoder-Decoder Architecture.
Encoder: mapping $(\mathbf{x}^1, .., \mathbf{x}^T) \mapsto \mathbf{z}$ (e.g. using a RNN, where $\mathbf{z} = \mathbf{h}^T$)
Decoder: mapping $\mathbf{z} \mapsto (\mathbf{y}^1, .., \mathbf{y}^S)$ (e.g. using a RNN with output feedback and a stop symbol).

12. Attention

Softmax Gating Function

$f_\phi(\xi, (\mathbf{h}_e^i, ..., \mathbf{h}_e^s)) = \frac{1}{\sum_j \exp[\phi(\xi, \mathbf{h}_e^j)]} \begin{pmatrix} \exp[\phi(\xi, \mathbf{h}_e^1)] \\ \cdot \\ \cdot \\ \cdot \\ \exp[\phi(\xi, \mathbf{h}_e^s)] \end{pmatrix}$
query vector $\xi \in \mathbb{R}^n$, set of values $\mathbf{h}_e^t \in \mathbb{R}^m$ $(t = 1, ..., s)$

Self-Gated Attention

Attention-based transfer function for sequences
$F(\xi, (\mathbf{h}_e^i, ..., \mathbf{h}_e^s)) = [\mathbf{h}_e^i \mathbf{h}_e^i ... \mathbf{h}_e^s] \cdot f_\phi(\xi, (\mathbf{h}_e^i, ..., \mathbf{h}_e^s))$
$\mathbb{R}^n = \mathbb{R}^{n \times s} \cdot \mathbb{R}^s$ applies attention key to every input; computes convex combination weights; convexly re-combines vectors

Seq2Seq with Attention

Attend to the **hidden state** sequence $(\mathbf{h}_e^1, ..., \mathbf{h}_e^2)$ of the encoding RNN.
Decoding RNN produces query at each time, i.e. $(\xi^1, ..., \xi^t)$.
Self-gated attention produces "read-outs" $(\mathbf{z}^1, ..., \mathbf{z}^t)$
Used as input to the decoding RNN: $(\mathbf{h}_d^r, \mathbf{z}^r) \mapsto \mathbf{y}_d^{t+1}$
Variant: use of bi-directional encoder RNNs.

Discussion

Feedforward network: unidirectional flow of information.
Recurrent networks: unrolling as feedforward network.
Memory units: persisting information, more efficient memorization, multiplicative gating.
Attention: learn to index, multiplicative gating to combine bottom-up and top-down information.

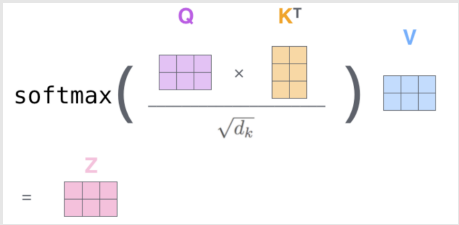
13. Transformer

Transformer

It is still an encoder-decoder architecture. More precisely, a stack of encoders and decoders. Final encoder fed to all decoder layers.
Structures: 1) Self-attention: to look at other words in the sentence; 2) Feed-forward: applied independently to each position; 3) Encoder-decoder attention layer in the Decoder, to focus on part of the input sentence.
Between the Self-Attention and the FFN, we have a normalisation layer that receives a residual connection.
$\text{LayerNorm}(x + \text{SubLayer}(x))$

Self-head Attention

Every input vector $\mathbf{x}_i$ is used in three ways in self-attention:
Query: compare $\mathbf{x}_i$ to every other vector to compute attention weights for its own output $\mathbf{y}_i$.
Key: compare $\mathbf{x}_i$ to every other vector to compute attention weights for the other outputs $\mathbf{y}_j$.
Value: use $\mathbf{x}_i$ in the weighted sum to compute every output vector based on these weights.
$\mathbf{q}_i = W_q \mathbf{x}_i$, $\mathbf{k}_i = W_k \mathbf{x}_i$, $\mathbf{v}_i = W_i \mathbf{x}_i$
$w'_{ij} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$, $\mathbf{w}_i = \text{softmax}(\mathbf{w}'_i)$, $z_i = \sum_j w_{ij} \mathbf{v}_j$



Multi-head Attention

Extend the concept of self-attention to multi-head attention: The standard Transformer implementation uses 8 attention heads, with randomly initialized weights. The attention can be computed in parallel.
Finally concatenate the results from each attention head $(Z_1, ..., Z_h)$ and transform this using another matrix of parameters W^o:
$\text{MultiHead}(Q, K, V) = \text{concat}(\text{head}_1, ..., \text{head}_h) W^o$,
where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

Position-wise FFN

The parameters are the same across different positions, but change from layer to layer:
$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$

Positional Encoding

We need to account for the order of the words, which is something implicit in sequential networks.
Solved by adding a vector to each input embedding. The pattern of this positional encoders is learnt by the model, which uses then to assess positions or difference in positions.

Encoder-decoder attention

1. **Encoder self-attention**: each position in the encoder can attend to all positions in the previous layer of the encoder, i.e. input to input attention.

2. **Decoder self-attention**: similar mechanism, but at position i we can only attend to known outputs at positions less than i, output to output attention. (By masking out invalid positions setting them to -Inf). **Queries** come from the previous decoder layer. But the **keys** and **values** from the output of the encoder.

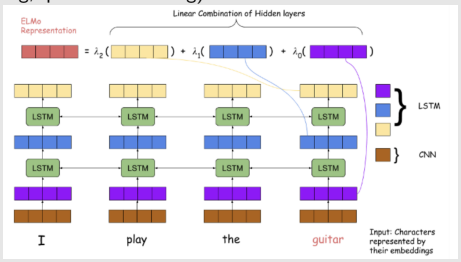
Transformer Conclusion

Type	Complexity	Sequence	Length
SA	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
RNN	$O(n \cdot d^2)$	$O(n)$	$O(n)$
CNN	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k n))$
restricted SA	$O(k \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Adam optimizer, with increasing learning rate during the first steps.

ELMO - Embedding from Language Models

Character-based model: morphology, OOV.
Left-to-right stacked LSTM: language model.
Right-to-left stacked LSTM: reverse language model.
Stacked LSTM layers, shared character embeddings and output embeddings.
Final representation: collapse all layers 2L + 1 linearly (Lower layer is better for Part-of-speech tagging, syntactic dependencies/ Higher layer is better for Sentiment, Semantic role labeling, question answering)



BERT - Bidirectional Encoder Representations from Transformers

Only uses the encoder stack of Transformers.
Most Relevant Contributions: 1) A mask based training approach. 2) A token for whole sentence embedding used for classifications tasks [CLS]. 3) A separator token [SEP]. 4) Ready for many downstream tasks.

Language Models

Large Language Models are few shot learners.
Training set: extremely massive, to meta-train the model. **Support set**: for the downstream task. **K-way**: the support set has k classes. **N-shot**: each class has n samples.
Multimodal few-shot learning with frozen LMs: Training is performed on single image-text pairs, and only updating the visual encoder weights.
(-): Achieve the necessary capacities to some degree; Far from SOTA performance.
(+): Frozen is able of open-ended interpretation of images and few-shot learning despite only trained to do captioning; Knowledge from LMs transfer to non-linguistic tasks; Starting point for research of multimodal few-shot learning; Many possible directions: more complex architectures, further few-shot training.

14. Variational Autoencoders

Linear autoencoder

Fully unsupervised approach: z acts as a bottleneck layer
low rank approximation Reduced SVD

Factor Analysis

Latent Variable Models: $p(z)$, $p(x|z)$, Mixture Models $z \in \{1, 2, ..., K\}$, $p(z)$ = mixing proportions, $p(x|z)$: conditional densities (e.g. Gaussians).
Linear Factor Analysis: Define a prior $z \sim \mathcal{N}(0, I)$. Linear observation model: x = μ + Wz + η, $\eta \sim \mathcal{N}(0, \Sigma)$, Σ is diagonal matrix s.t. $\Sigma := \text{diag}(\sigma_1^2, ..., \sigma_n^2)$. 1) η and z is independent 2) $m < n$ fewer factors than features 3) few factors account for dependencies b/w many observables 4) MLE for μ on training set, $\hat{\mu} = \frac{1}{t}\sum_{t=1}^{s} \mathbf{x}_t$(centering).
$x \sim \mathcal{N}(\mu, \mathbf{W}\mathbf{W}^\top + \Sigma)$
Moment Generating Functions
Moment generating function of random vector $\mathbf{x}$, $M_\mathbf{x} : \mathbb{R}^n \to \mathbb{R}$, $M_\mathbf{x}(\mathbf{t}) := \mathbf{E}_\mathbf{x}\exp[\mathbf{t} \cdot \mathbf{x}]$
Uniqueness theorem: If $M_\mathbf{x}$, $M_\mathbf{y}$ exist for RVs $\mathbf{x}$, $\mathbf{y}$ and $M_\mathbf{x} = M_\mathbf{y}$ then (essentially) $p(\mathbf{x}) = p(\mathbf{y})$, $M_{\mathbf{x}+\mathbf{y}} = M_{\mathbf{x}\mathbf{y}}$.
For multivariate Gaussian, $M_\mathbf{x}(\mathbf{t}) = \exp[\mu \cdot \mathbf{t} + \frac{1}{2}t^\top \Sigma t]$
Apply Generating Functions to Linear Factors
define $\tilde{x} := \mathbf{W}\mathbf{z}$ such that $\mathbf{x} = \tilde{x} + \mu + \eta$. $M_x(\mathbf{t}) = M_\mathbf{z}\mathbf{W}^\top\mathbf{t} = \exp[\frac{1}{2}\mathbf{t}^\top(\mathbf{W}\mathbf{W}^\top)\mathbf{t}]$. $M_\mathbf{x} = \exp[\mu \cdot \mathbf{t} + \frac{1}{2}\mathbf{t}^\top(\mathbf{W}\mathbf{W}^\top + \Sigma)\mathbf{t}]$. factors are only identifiable up to rotations, reflections, or permutations.

Latent Variable Models

DeFinetti's Theorem: exchangeable data, we can decompose them by a latent variable model.
EM Algorithm: ELBO. $\log p_\theta(\mathbf{x}) \geqslant \mathbf{E}_q[\log p_\theta(\mathbf{x}, \mathbf{z})] + \mathbf{H}(q(\mathbf{z})) = \mathbf{E}_q \log p_\theta(\mathbf{x}|\mathbf{z}) - \mathbf{KL}(q(\mathbf{z})||p_\theta(\mathbf{z}))$. Variational inference: further approximation, restrict to simpler families of distribution, amortized inference → variational autoencoders.
Implicit Models: Statistical models via: generating stochastic mechanism or simulation process.

Variational Autoencoders

Use a neural network for generative modeling. Posterior non-tractable, approximate by another NN.

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) = D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})||p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}^{(i)})) + \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)})$$
$$\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) \geq \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[-\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) + \log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})]$$
$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)}) = -D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})||p_{\boldsymbol{\theta}}(\mathbf{z})) + \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z})\right]$$

Inference Network mapping can be realized by independent (deep) network, parametric form with parameters ϕ: generalization across x, aka amortized inference. KL-divergence can be thought of as regularization. 1) assume that for given x, the encoder is fixed. Sample noise variables $\mathbf{z} \sim \mathcal{N}(\mu(x), \Sigma(x))$
2) Optimizing over q involves gradients of expectations.
3) Stochastic back-propagation (re-parameterize z) $\mathbf{z} \sim \mathcal{N}(\mu, \Sigma)$ is equivalent to $\mathbf{z} = \mu + \Sigma^{\frac{1}{2}}\eta$, $\eta \sim (\mathbf{0}, \mathbf{I})$ (can now back propagate w.r.t. μ and $\Sigma^{\frac{1}{2}}$).
Differentiation with respect to θ and ϕ. Naive Monte Carlo Estimate yields high variance estimators. repar: substitute $\tilde{z}$ by $g_\phi(\epsilon, x)$. Now:

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})}[f(\mathbf{z})] = \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[f(g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}, \mathbf{x}^{(i)}))\right] \simeq \frac{1}{L}\sum_{l=1}^{\tilde{L}} f(g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}^{(l)}, \mathbf{x}^{(i)}))$$

leading to General Stochastic Gradient Variational Bayes:
$\tilde{\mathcal{L}}^A(\theta, \phi; \mathbf{x}^{(i)}) =$
$\frac{1}{L}\sum_{l=1}^{L} \log p_\theta(\mathbf{x}^{(i)}, \mathbf{z}^{(i,l)}) - \log q_\phi(\mathbf{z}^{(i,l)}|\mathbf{x}^{(i)})$, with $\mathbf{z}^{(i,l)} = g_\phi(\epsilon^{(i,l)}, \mathbf{x}^{(i)})$ and $\epsilon^{(l)} \sim \mathbf{p}(\epsilon)$

15. Adversarial Robustness

Robust

A trained model is said to be **robust** if (at test time) it returns correct output on all small manipulations of its input data. Robustness to common corruptions is indeed a desired property for DL models. Adversarial examples can be seen as worst-case corruptions, as opposed to random or typical corruptions. Adversarial examples can be used to reveal hidden biases.
Spatial adversarial examples: 1) Geometric transformations; 2) Spatial transformations.

Adversarial Perturbations

Let $k : \mathbb{R}^d \to \{1, 2, .., C\}$ be a C-class classifier, and $x \in \mathbb{R}^d$ be an input. We define an adversarial perturbation for x as: Find $r \in \mathbb{R}^d$ such that $k(x + r) \neq k(x)$, and $x + r$ is similar to $x(||r||_p$ should be small).
Measuring robustness to adversarial perturbations:
1. robustness can be measured as the average norm of "minimal adversarial perturbations".
We want to solve: $\underset{r}{\text{argmin}}||r||_p$ s.t. $k(x + r) \neq k(x)$
DeepFool: Since we have a closed-form solution for linear classifiers, we will solve it by solving a sequence of linearized problems. (E.g. binary case: $\underset{r}{\text{argmin}}||r||_p$ s.t. $(\nabla_x f_1(x) - \nabla_x f_2(x))^\top r + f_1(x) - f_2(x) < 0)$
In Action (Iterative):
1) $\underset{r_1}{\text{argmin}}||r_1||_2$ s.t. $(\nabla_x f_1(x) - \nabla_x f_2(x))^\top r_1 + f_1(x) - f_2(x) < 0 \to x_1 = x + r_1$
2) $\underset{r_2}{\text{argmin}}||r_2||_2$ s.t. $(\nabla_x f_1(x) - \nabla_x f_2(x))^\top r_2 + f_1(x) - f_2(x) < 0 \to x_2 = x + r_1 + r_2$
Pros: (almost) Parameter-free, fast yet accurate on deep models.
Cons: Poor performance on kernel classifiers, no obvious way to generalize to non-classification tasks.
2. given $\epsilon \geqslant 0$, one can define robustness as the proportion of input samples for which there exist no adversarial perturbation with $||r||_p \leqslant \epsilon$.

Robust Deep Learning

Adversarial training Instead of training on the original data, train on the adversarial examples.
Standard training loss: $\min_\theta \text{loss}(f_\theta(x), y)$
Robust training loss: $\min_\theta \max_{||r||_p \leqslant \epsilon} \text{loss}(f_\theta(x + r), y)$
Alternating scheme We alternatively minimise, w.r.t. parameters, and maximize, w.r.t. perturbation.
Maximisation step given current parameters θ^t, we compute perturbation r^{t+1}:
$r^{t+1} = \underset{||r||_p \leqslant \epsilon}{\text{argmax}} \text{loss}(f_{\theta^t}(x + r), y)$
Minimisation step given current parameters r^{t+1}, we update the parameters:
$\theta^{t+1} = \theta^t - \eta\nabla_\theta \text{loss}(f_{\theta^t}(x + r^{t+1}), y)$
Adversarial training Instead of training on the original data, train on the most difficult samples, a.k.a. adversarial examples.
To solve the max step: **Projected gradient ascent**
Given the network's parameters θ, for $p = \infty$,
$\tilde{r}^{s+1} \leftarrow r^s + \alpha\text{sign}(\nabla_x \text{loss}(f_\theta(x + r^s), y))$
$r^{s+1} \leftarrow \prod_\epsilon^\infty(\tilde{r}^{s+1})$
fast gradient sign method (FGSM): 1-iteration l_∞-PGD: $r = \epsilon\text{sign}(\nabla_x \text{loss}(f_\theta(x), y))$

16. GAN

GAN

Classification: distinguish between data & model:
$\tilde{p}_\theta(\mathbf{x}, y) = \frac{1}{2}(yp(\mathbf{x}) + (1 - y)p_\theta(\mathbf{x}))$, $y \in \{0, 1\}$
Train generator via minimizing the logistic likelihood:
$\theta \overset{\min}{\to} l^*(\theta) := \mathbf{E}_{\tilde{p}_\theta}[y\ln q_\theta(\mathbf{x}) + (1 - y)\ln(1 - q_\theta(\mathbf{x}))]$
Objective tries to recover the **Jensen-Shannon divergence**.
Evaluating GANs How to measure quality of implicit models: isual inspection (inception score). Trade-offs: 1. noisy samples (e.g. blurry images), but adequate representation of the variability. 2. faithful (as in good looking) samples, but lack of representation ("mode dropping").
$F(\theta, w) = \mathbb{E}_{x\sim P}[log(D_w(x))] + \mathbb{E}_{x\sim Q_\theta}[log(1 - D_w(x))]$,
P(x) is the data distri, Q_θ is the model.
$\text{argmin}_G \text{max}_D \mathbb{E}_{z,x}[\log D(x) + \log(1 - D(G(z)))]$

Evaluating GAN

convergence to $p(\mathbf{x})$. Out-of-sample evaluations is not available for implicit models. Trade-offs: noisy samples/faithful samples.

Normalizing Flows

Bijections F which are convenient to compute, invert and calculate $|\det(\partial F)|$
Compositionality: $\det(\partial F) = \prod_{l=L}^{1} \det(\partial F_l \circ \partial F_{l-1:1})$, $\det(\partial F^{-1}) = \det(\partial F)^{-1}$
Log-likelihood: $\log p(x|z) = -\sum_{l=1}^{L} \log|\det(\partial F_l \circ \partial F_{l-1:1})|$
Linear Flow: Requires $\mathbf{O}(n^3)$ to compute determinant and inverse. Normalizing flows are often not powerful enough as (unconditional) generative models on their own (Jacobian condition too restrictive).

AutoRegressive DNNs

Autoregressive models - generate output one variable at a time.
PixelCNN: network models conditional distribution of every individual pixel given previous pixels. Drawback: Slow process.

Method	Latent	Density	Generator
Autoregressive models	no	yes	slow
Diffusion Models	(high-dim)	lower-bound	slow
GANs	yes	no	yes
VAEs	yes	lower-bound	yes
Energy-based methods	no	unnormalized	no